P, NP, NP-Complete & NP-Hard Problems

Dr. Malay S. Bhatt
Associate Professor
Department of Computer Engineering
Dharmsinh Desai University

Introduction

We can solve many problems **efficiently** because we have available (and choose to use) **efficient algorithms.**

Given any problem for which you know ***some*** algorithm, it is always possible to write an **inefficient** algorithm to "solve" the problem.

For example, consider a sorting algorithm that tests every possible permutation of its input until it finds the correct permutation that provides a sorted list. The running time for this algorithm would be unacceptably high, because it is proportional to the number of permutations which is n! for n inputs.

Introduction

When solving the ***minimum-cost spanning tree problem***, if we were to test every possible subset of edges to see which forms the shortest minimum spanning tree, the amount of work would be proportional to $2^{|E|}$ for a graph with |E| edges.

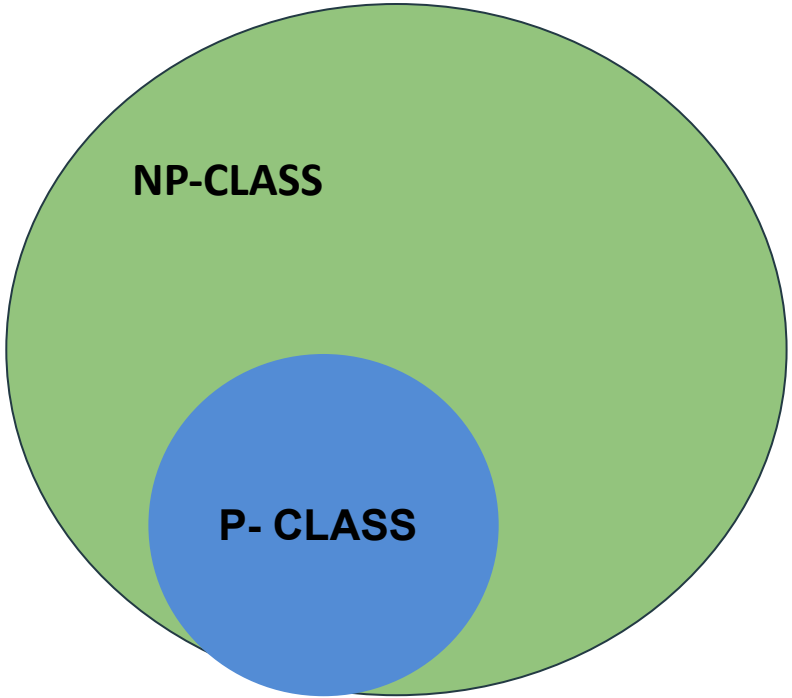
Fortunately, for both of these problems we have more clever algorithms that allow us to find answers (relatively) quickly without explicitly testing every possible solution.

P Class

By now you have studied many data structures that can be used in a wide variety of problems, and many examples of efficient algorithms. In general, our **search algorithms** strive to be in **O(logn)** to find a record, and our **sorting algorithms** strive to be in **O(nlogn).**

You might have come across a few algorithms have higher asymptotic complexity. Standard matrix multiplication algorithm have running times of $\Theta(n^3)$

P and NP Class



NP Class

Imagine a magical computer that works by **guessing** the correct solution from among all of the possible solutions to a problem.

Another way to look at this is to **imagine a super parallel computer** that could test all possible solutions simultaneously. Certainly this magical (or highly parallel) computer can do anything a normal computer can do. It might also solve some problems more quickly than a normal computer can.

NP- Algorithm Examples

```
NP_Linear_Search(A,x)
{
j=Choice(1,n)
If (A[j]==x)
{
      write(j)
      Success()
}
Write(0)
Failure()
}
```

Time Complexity is O(1) for Non-Deterministic Linear Search, while it is O(n) for Deterministic Linear Search.

NP- Algorithm Examples

```
NP_Decison_Knapsack(p, w, n, m, r, x)
{
W=0
P=0
For i = 1 to n
{
x[i]=Choice(0,1)
W = W+w[i]*x[i]
P =  P+p[i]*x[i]
}
if((W>m ) or (P<r)) Failure()
Else        Success()
}
```

Time Complexity is O(n) for Non-Deterministic Algorithm.

P and NP Class

Consider some problem where, given a guess for a solution, **verifying** the solution to see if it is correct can be done in **polynomial time**. Even if the number of possible solutions is exponential, any given guess can be checked in polynomial time (equivalently, all possible solutions are checked simultaneously in polynomial time), and thus the problem can be solved in polynomial time by our hypothetical magical computer.

Another view of this concept is this: If you cannot get the answer to a problem in polynomial time by guessing the right answer and then checking it, then you cannot do it in polynomial time in any other way.

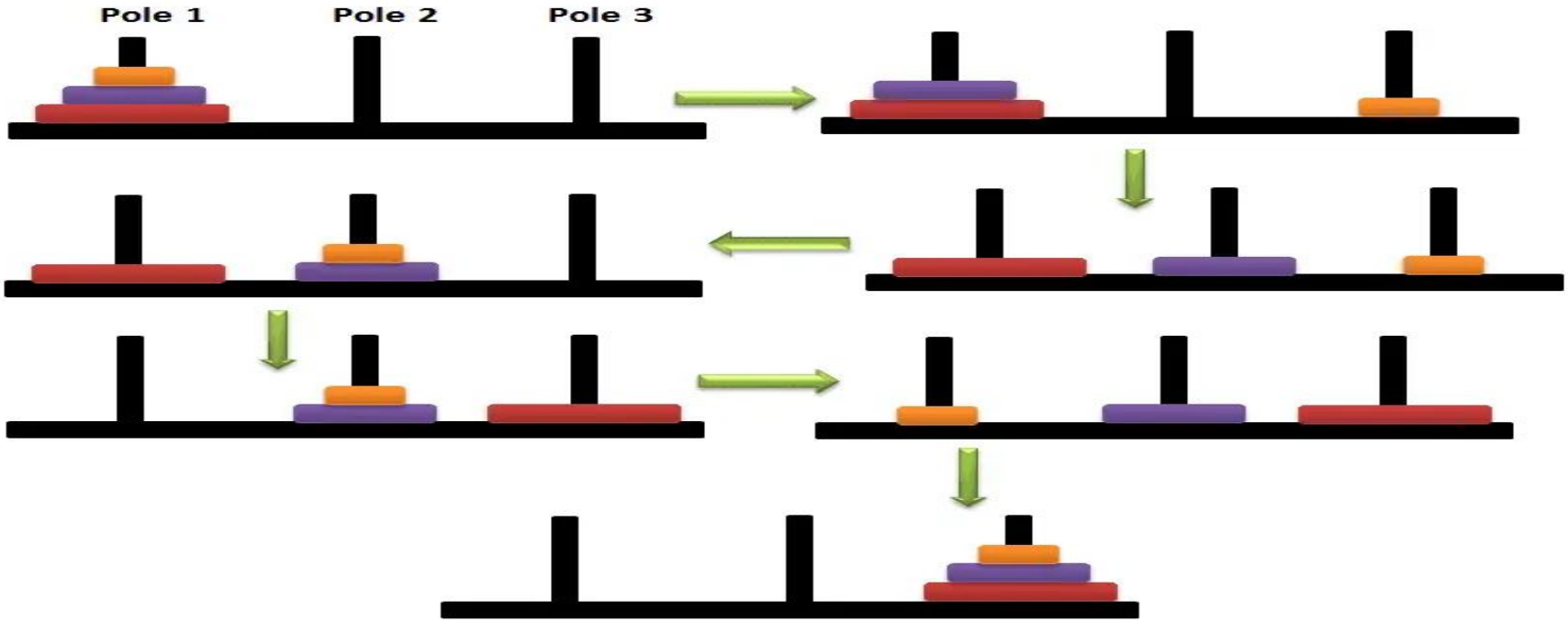
Non-Deterministic Algorithm

The idea of "guessing" the right answer to a problem—or checking ***all possible solutions*** in ***parallel*** to determine which is correct—is a called a ***non-deterministic choice***. An algorithm that works in this manner is called a ***non-deterministic algorithm***, and any problem with an algorithm that runs on a non-deterministic machine in polynomial time is given a special name: It is said to be a problem in NP.

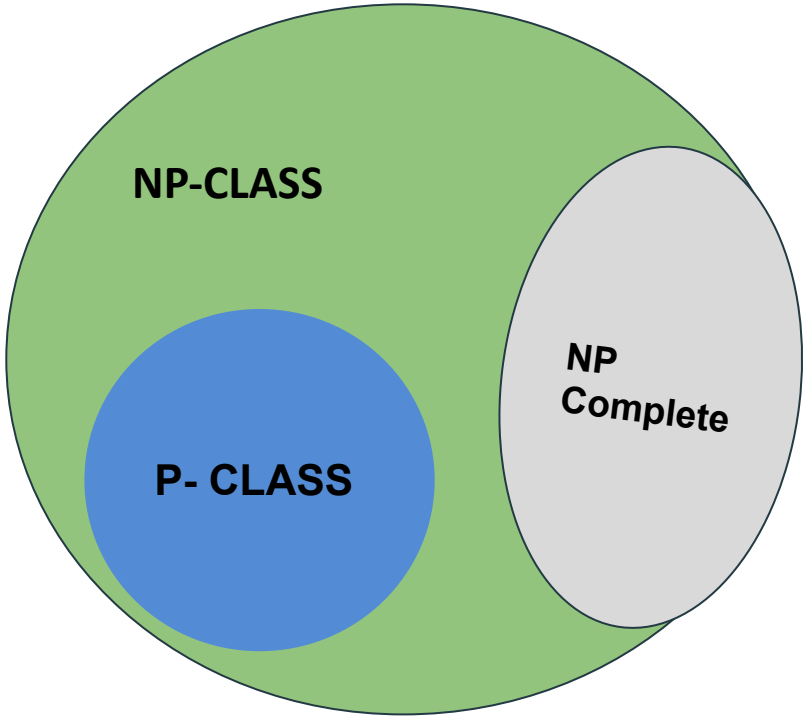
Thus, problems in NP are those problems that can be solved in polynomial time on a **non-deterministic machine (Non-Deterministic Turing Machine)**.

Not all problems requiring exponential time on a regular computer are in NP. For example, **Towers of Hanoi is *not* in NP**, because it must print out $O(2^n)$ moves for n disks. A non-deterministic machine cannot "guess" and print the correct answer in less time.

Tower of Hanoi



P, NP & NP -Complete Class



P and NP Class

There is a large collection of problems with following **property**: We **know efficient non-deterministic algorithms**, but we do not know if there are efficient deterministic algorithms. At the same time, we have not been able to prove that any of these problems do *not* have efficient deterministic algorithms. This class of problems is called ***NP-complete***.

What is truly strange and fascinating about NP-complete problems is that if anybody ever finds the solution to any one of them that runs in polynomial time on a regular computer, then by a series of **reductions**, every other problem that is in NP can also be solved in polynomial time on a regular computer!

A problem X is defined to be NP-complete if

1. X is in NP, and
2. if *any* problem in NP can be reduced to X in polynomial time **(i.e. X is NP-hard)** .

Relationships among P, NP, NP-Complete

Problems that are solvable in polynomial time on a regular computer are said to be in class P.

Clearly, all problems in P are solvable in polynomial time on a non-deterministic computer simply by neglecting to use the non-deterministic capability.

NP is the set of all problems solvable by nondeterministic algorithm in Polynomial time.

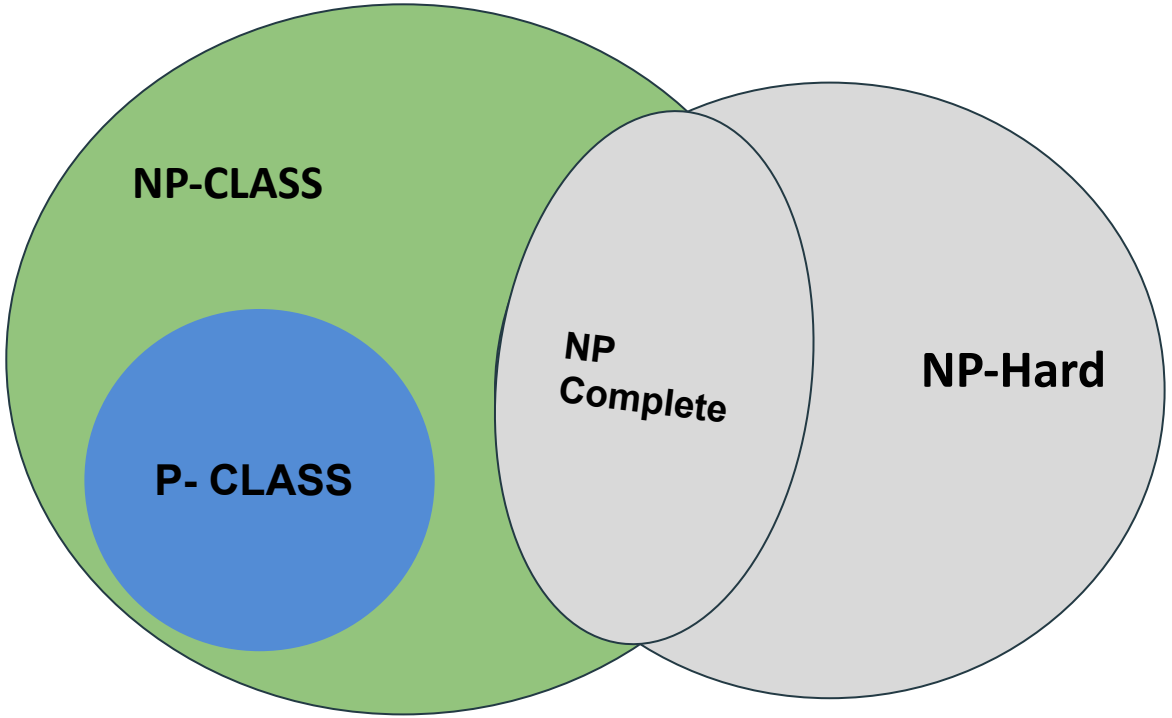
Some problems in NP are NP-complete.

Advantages of knowing that a problem is NP-Complete

The primary **theoretical advantage** of knowing that a problem P1 is NP-complete is that it can be used to show that another problem P2 is NP-complete. This is done by finding a **polynomial time reduction** of P1 to P2. Because we already know that all problems in NP can be reduced to P1 in polynomial time (by the definition of NP-complete), we now know that all problems can be reduced to P2 as well by the simple algorithm of reducing to P1 and then from there reducing to P2.

There is a **practical advantage** to knowing that a problem is NP-complete. It relates to knowing that if a deterministic polynomial time solution can be found for *any* problem that is NP-complete, then a deterministic polynomial solution can be found for *all* such problems.
The implication is that,
Because no one has yet found such a solution, it must be difficult or impossible to do; and
Effort to find a polynomial time solution for one NP-complete problem can be considered to have been expended for all NP-complete problems.

P, NP, NP -Complete & NP- Hard Classes



NP-Hard Problem

A Problem X is NP-Hard if there is an **NP-Complete problem Y,** such that **Y is reducible to X in polynomial time.** NP-Hard problems are **as hard as** NP-Complete problems.

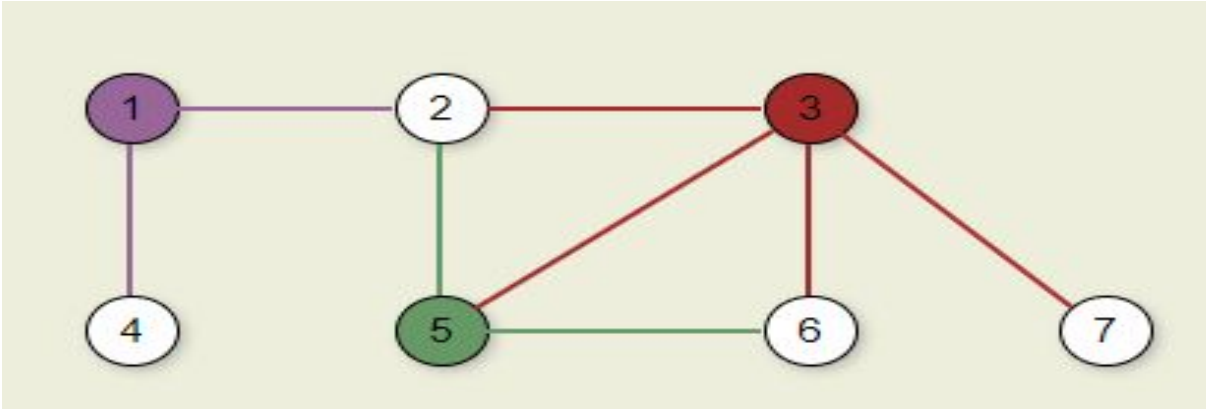
NP- Hard problem may be harder than NP- Complete Problem.

NP-Hard Problem need **not** be in NP class.

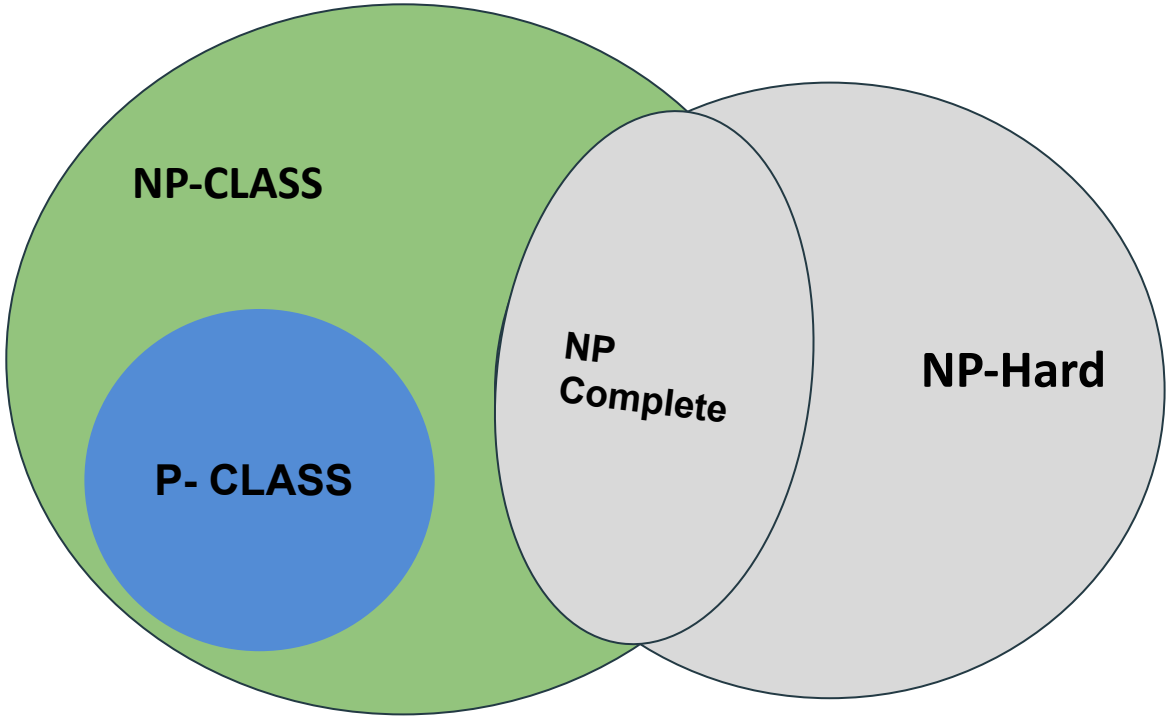
Vertex Cover Problem

In graph theory, a **vertex cover** (sometimes **node cover**) of a graph is a set of vertices that includes at least one endpoint of every edge of the graph.

Problem – Given a graph G(V, E) and a positive integer k, the problem is to find whether there is a subset V' of vertices of size at most k, such that every edge in the graph is connected to some vertex in V'.



Vertex Cover is NP- Complete



Vertex Cover is NP- Complete

We want to prove :

1. **Vertex Cover is in NP (Solution of instance is verifiable in Polynomial time)**
2. **Vertex cover is NP- Hard (Clique Problem reduces to Vertex Cover)**

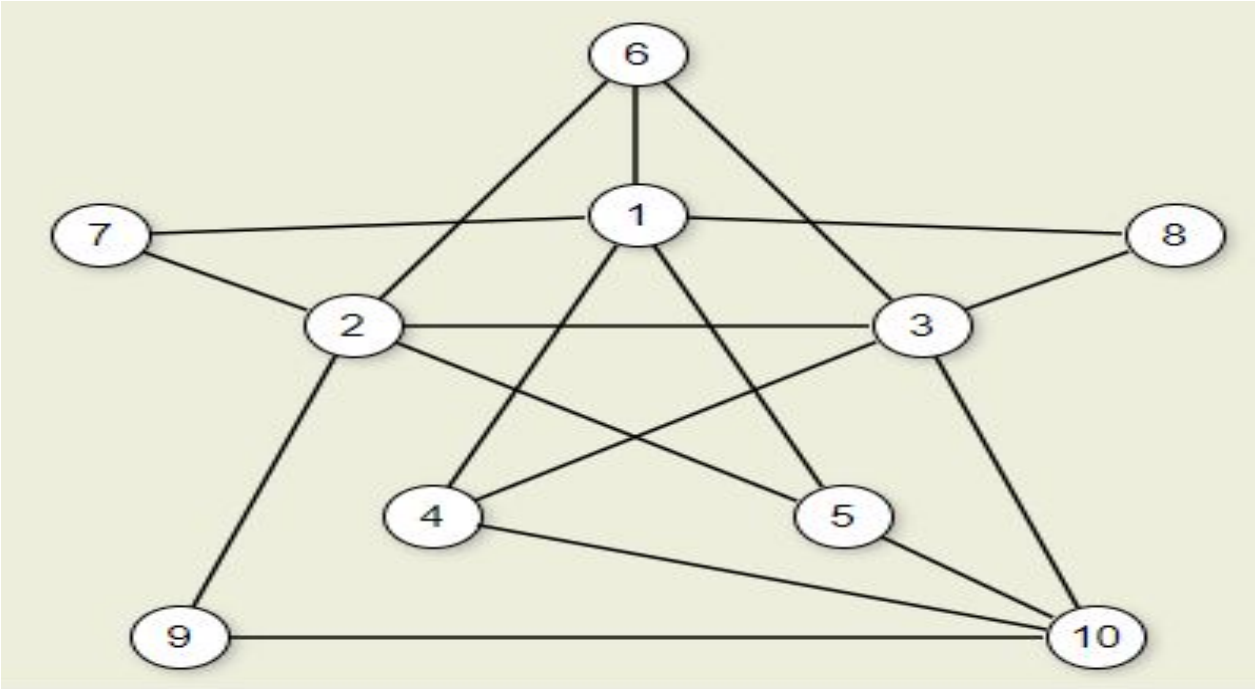
Vertex Cover is NP- Complete

Proof that vertex cover is in NP –
If any problem is in NP, then, given a ‘certificate’ (**a solution)** to the problem and an **instance of the problem** (a graph G and a positive integer k, in this case), we will be able to verify (check whether the solution given is correct or not) the certificate in **polynomial time.**

The certificate for the vertex cover problem is a subset V’ of V, which contains the vertices in the vertex cover. We can check whether the set V’ is a vertex cover of size k or not.

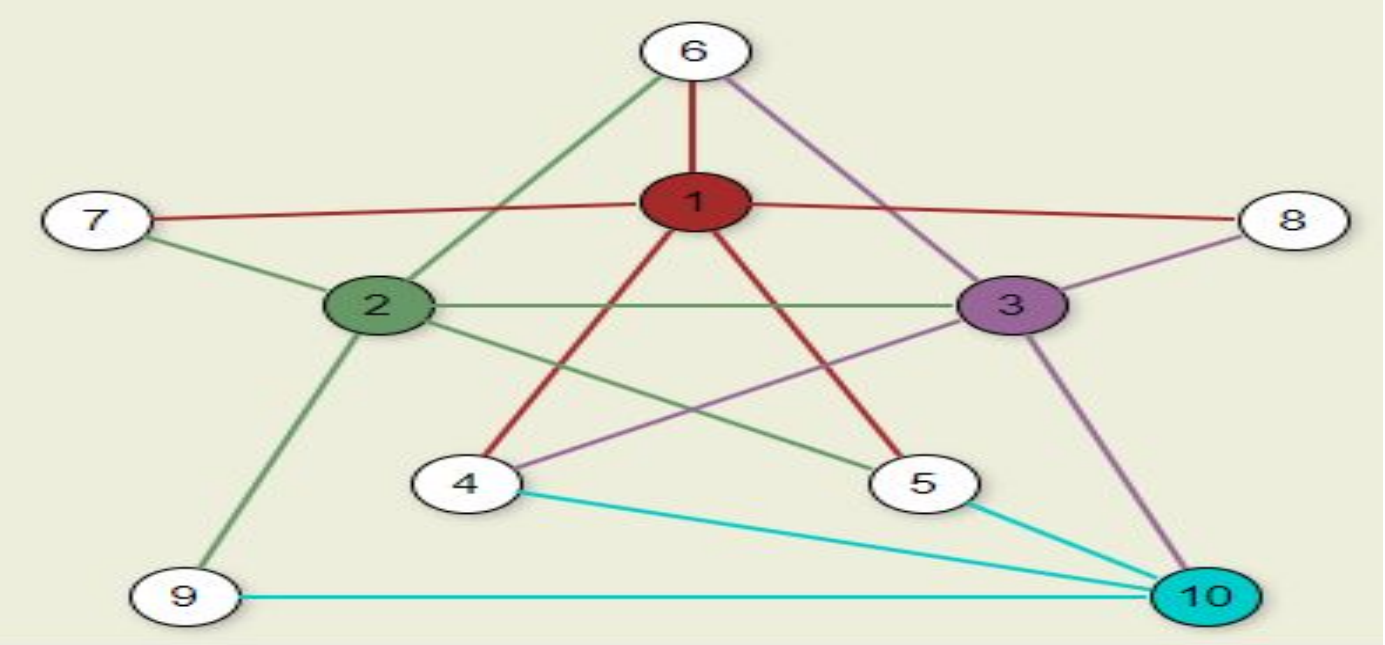
Vertex Cover Problem Instance

Does the graph given below has a vertex cover of size less than or equal to 4 ?



Vertex Cover Problem Instance Solution

Does the graph given below has a vertex cover of size less than or equal to 4 ?



Vertex Cover is NP- Complete

Verifying the solution in Polynomial time

```
let count be an integer
set count to 0
for each vertex v in V'
    remove all edges adjacent to v from set E
    increment count by 1
    if count = k and E is empty
    then
        the given solution is correct
    else
            the given solution is wrong
```

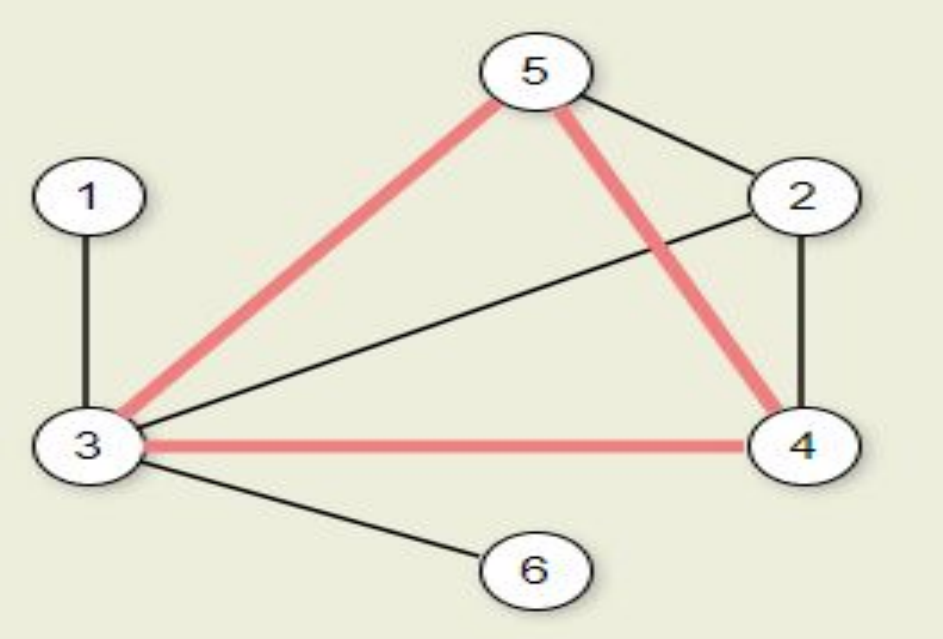
Vertex Cover is NP- Complete

Proof that vertex cover is NP Hard –
To prove that Vertex Cover is NP Hard, we take some problem which has already been proven to be NP-Complete, and show that this problem can be reduced to the Vertex Cover problem.

It has been proven that Max-Clique Problem is a NP-Hard Problem.
It has been proven that Clique Problem is a NP-Complete Problem

Clique Problem

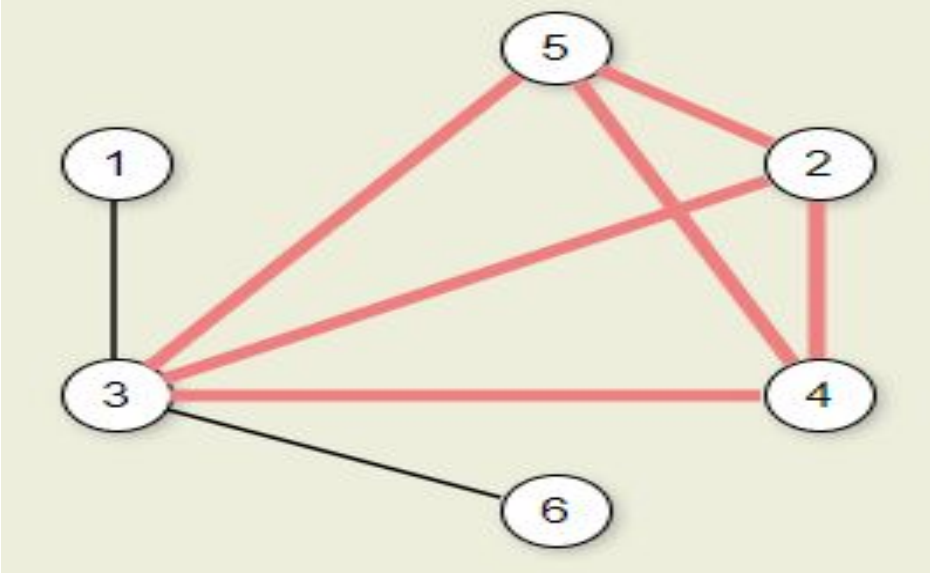
If in a graph G, there exists a **complete subgraph** of k nodes, then G is said to contain k-clique.



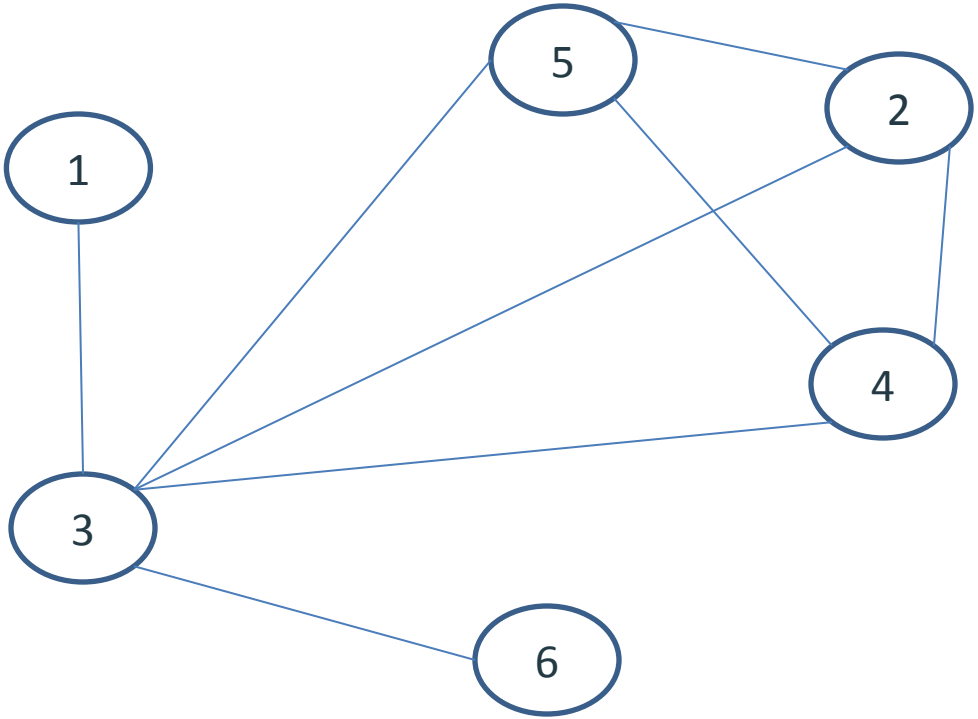
Max Clique Problem

If in a graph G, there exists a complete subgraph of k nodes, then G is said to contain k-clique.

Clique with largest number of vertices in a graph G is called **Maximum Clique** in G



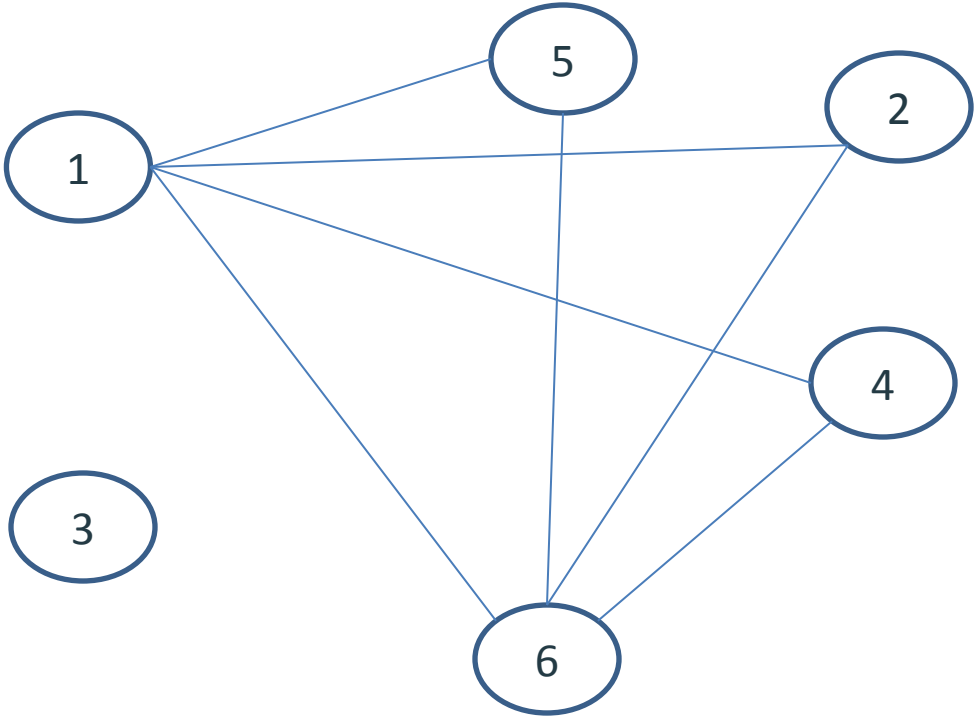
Clique $\leq_R$ Vertex Cover



Graph G

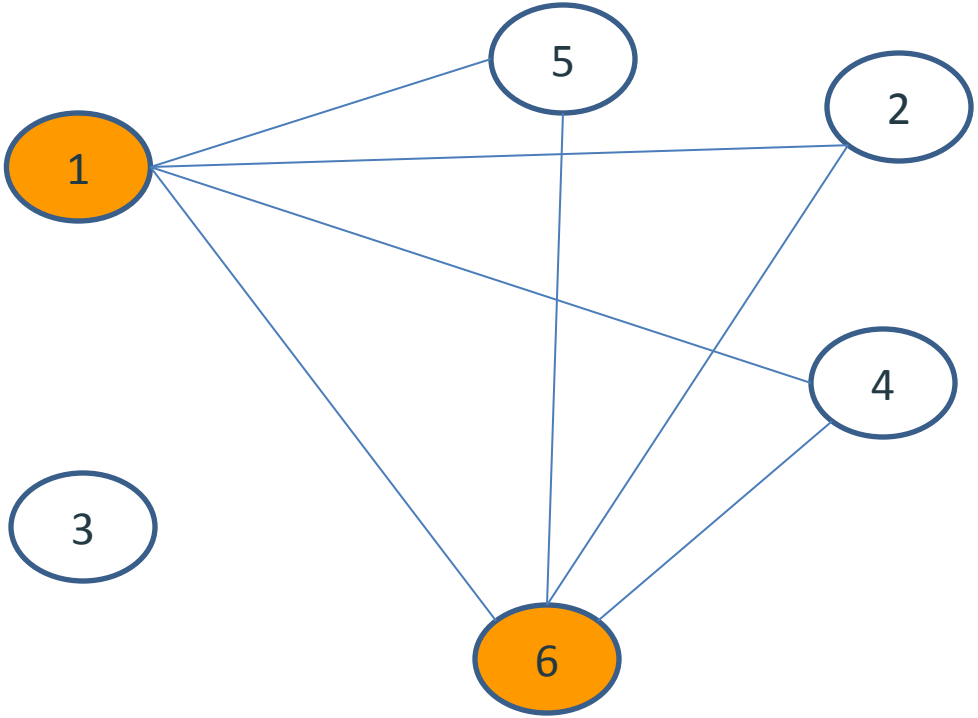
Clique $\leq_R$ Vertex Cover

Graph G’

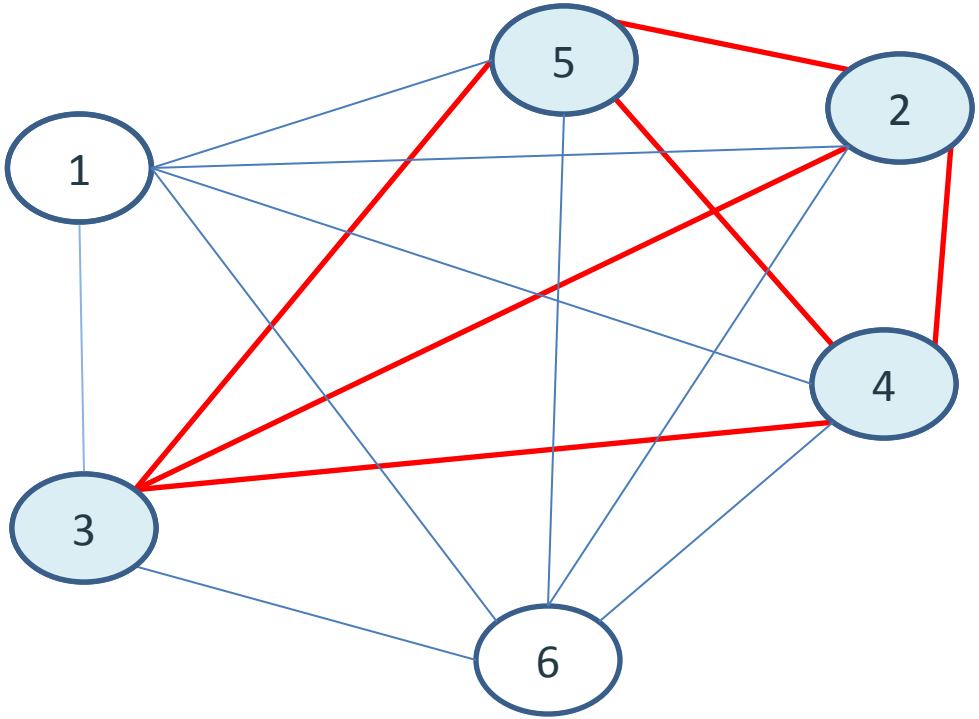


Clique $\leq_R$ Vertex Cover

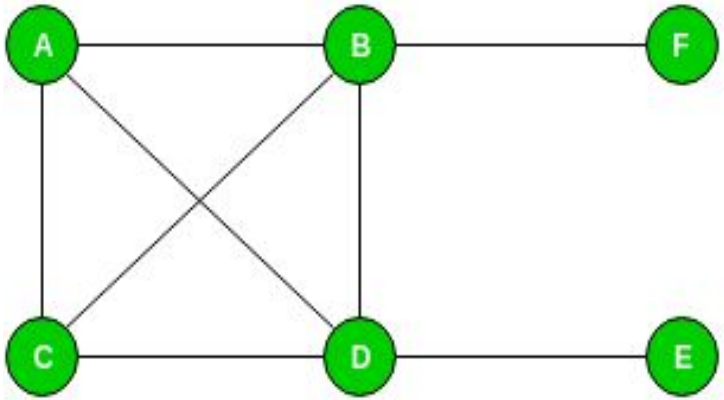
Graph G'



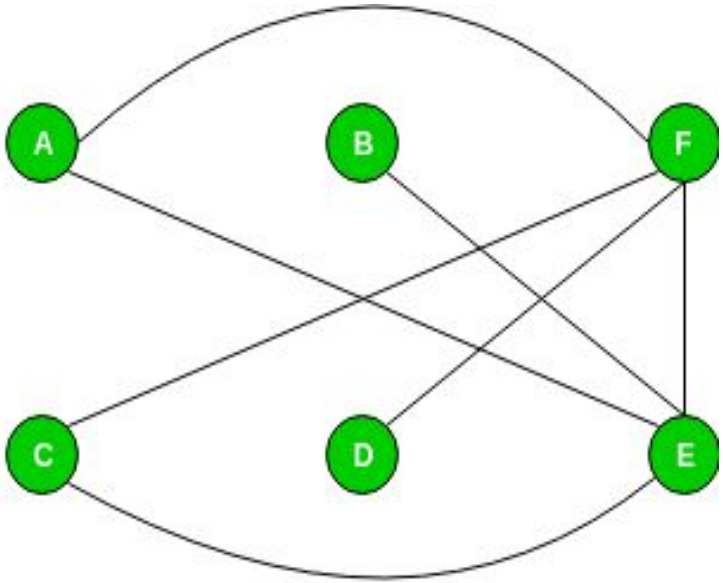
Clique $\leq_R$ Vertex Cover



Maximum Clique $\leq_R$ Vertex Cover (Example -2)



G
V' ={A, B, C, D}



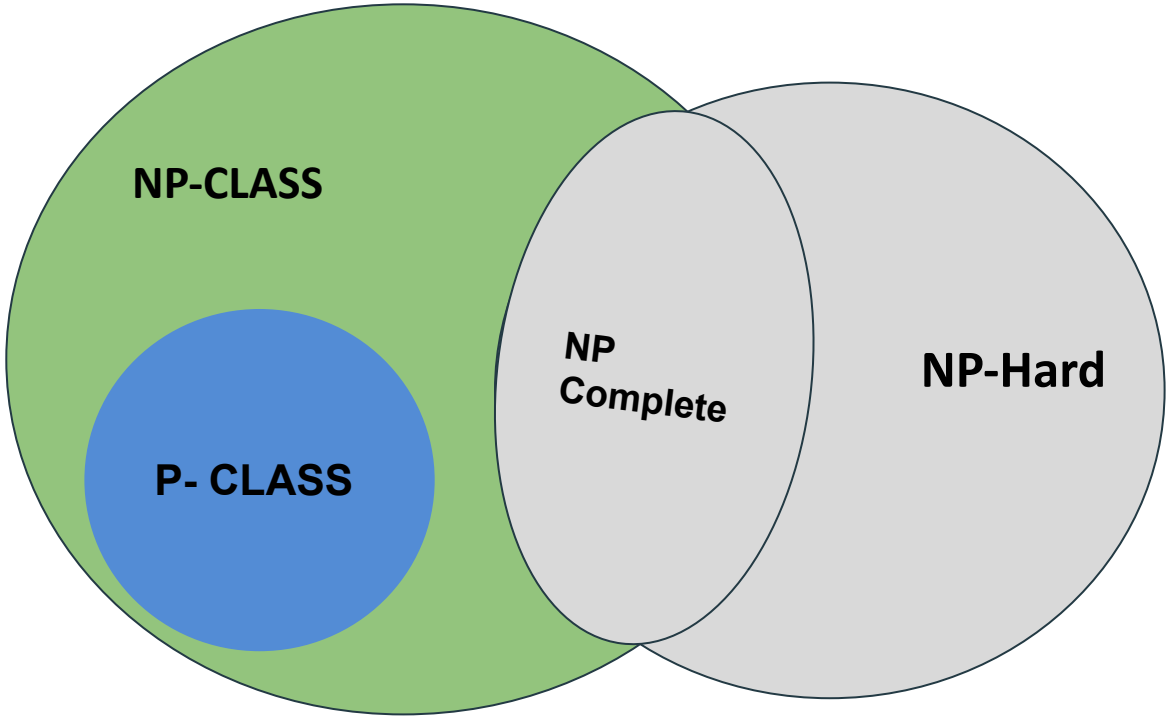
G'
V'' ={E, F}

Vertex Cover is NP- Complete

We have proved :

1. **Vertex Cover is in NP (Solution of instance is verifiable in Polynomial time)**
2. **Vertex cover is NP- Hard (Clique Problem reduces to Vertex Cover)**

Clique Decision Problem is NP- Complete



Clique Decision Problem is NP-Complete

We want to prove :

1. **Clique Decision Problems is in NP (Solution of instance is verifiable in Polynomial time)**
2. **Clique Decision Problem is NP- Hard (3-SAT reduces to Clique Decision Problem)**

Clique Decision Problem is NP-Complete

The Clique Decision Problem belongs to NP :

Let the certificate be a set S consisting of nodes in the clique and S is a subgraph of G. We have to check if there exists a clique of size k in the graph.

Hence, verifying if number of nodes in S equals k, takes O(1) time.

Verifying whether each vertex has an out-degree of (k-1) takes $O(k^2)$ time. (Since in a complete graph, each vertex is connected to every other vertex through an edge.).

Therefore, to check if the graph formed by the k nodes in S is complete or not, it takes $O(k^2) = O(n^2)$ time (since k<=n, where n is number of vertices in G).

Clique Decision Problem is NP-Complete

It has been proven that SAT Problem is a NP-Hard Problem.
(Cook's Theorem)

The Boolean Satisfiability Problem (SAT) is an NP-Complete problem as proved by the Cook's theorem.

Therefore, every problem in NP can be reduced to SAT in polynomial time. Thus, if SAT is reducible to Clique in polynomial time, every NP problem can be reduced to Clique in polynomial time, thereby proving Clique to be NP-Hard.

3-SAT $\leq_R$ K- Clique

Given a formula in Conjunctive Normal Form (CNF), 3-SAT problem is to find whether the formula is satisfiable.

For a given graph G=(V,E) and integer k, the Clique problem is to find whether G contains a clique of size >= k.

Let's consider the Formula φ = (x1+x2+x3') (x1'+x2' + x4)

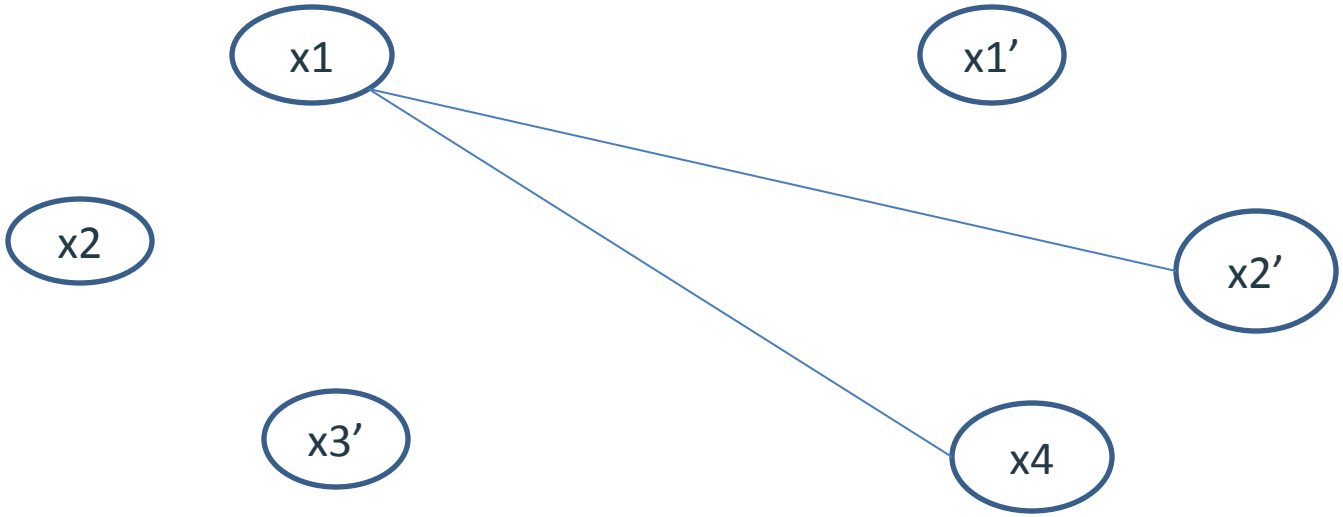
Reduction Procedure:
Construct a graph G of k clusters with 3 nodes in each cluster. Each cluster corresponds to a clause in **φ**.

Each node in a cluster is labeled with a literal from the clause.

Put an edge between all pairs of nodes in different clusters except for the pairs of the form (x, x'). No edge is put between any pair of nodes in the same cluster

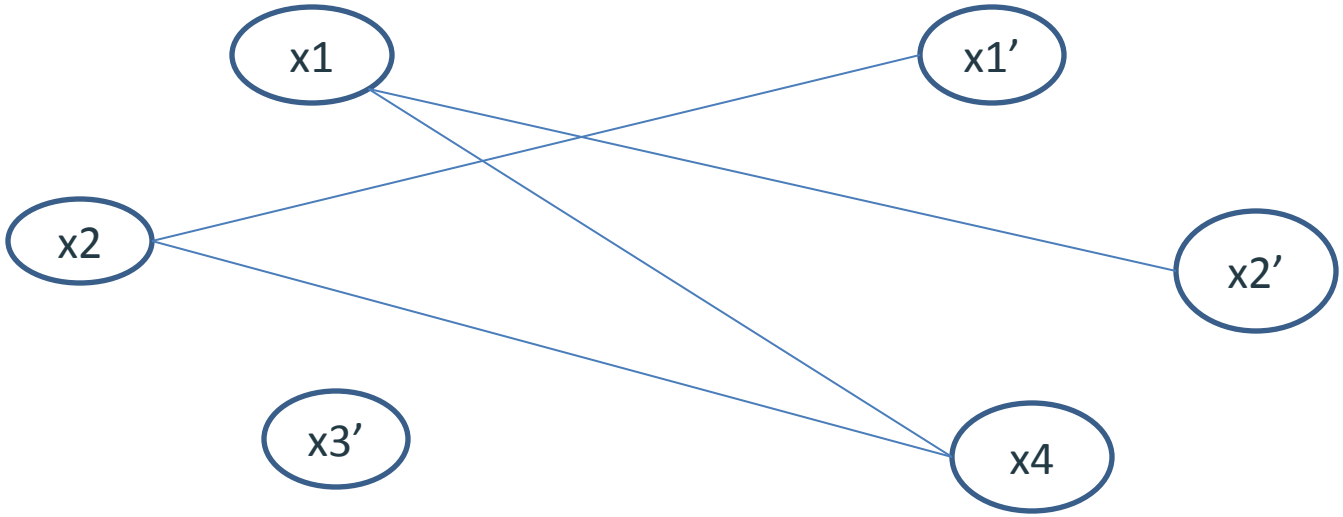
3-SAT $\leq_R$ K- Clique

Let's consider the Formula φ = (x1+x2+x3') (x1'+x2' + x4)



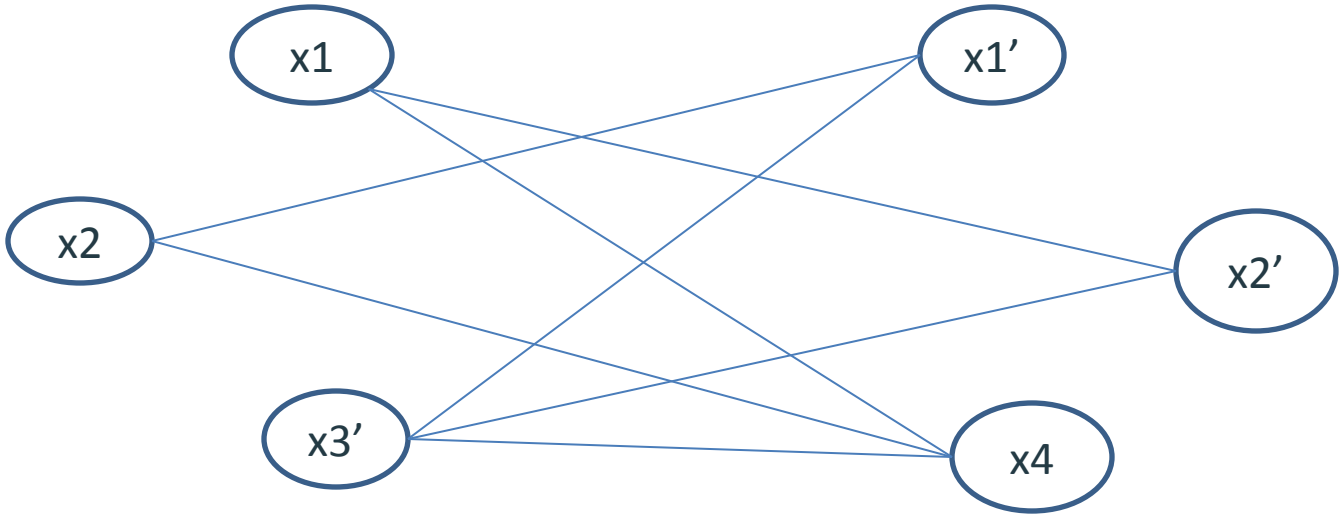
3-SAT $\leq_R$ K- Clique

Let's consider the Formula **φ** = (x1+x2+x3') (x1'+x2' + x4)



3-SAT $\leq_R$ K- Clique

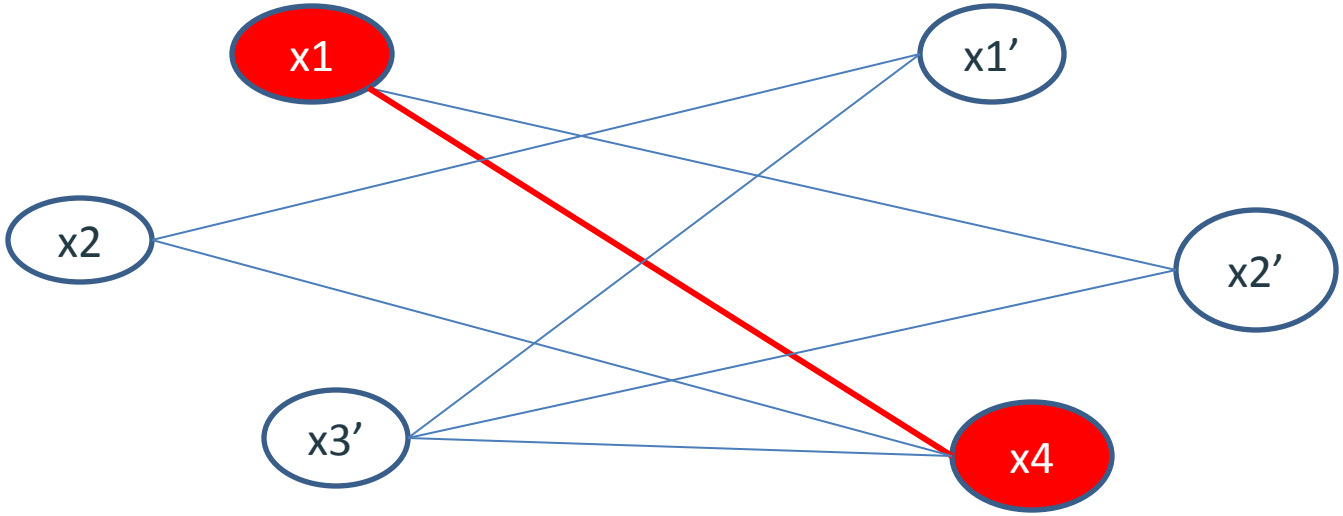
Let's consider the Formula φ = (x1+x2+x3') (x1'+x2' + x4)



3-SAT $\leq_R$ K- Clique

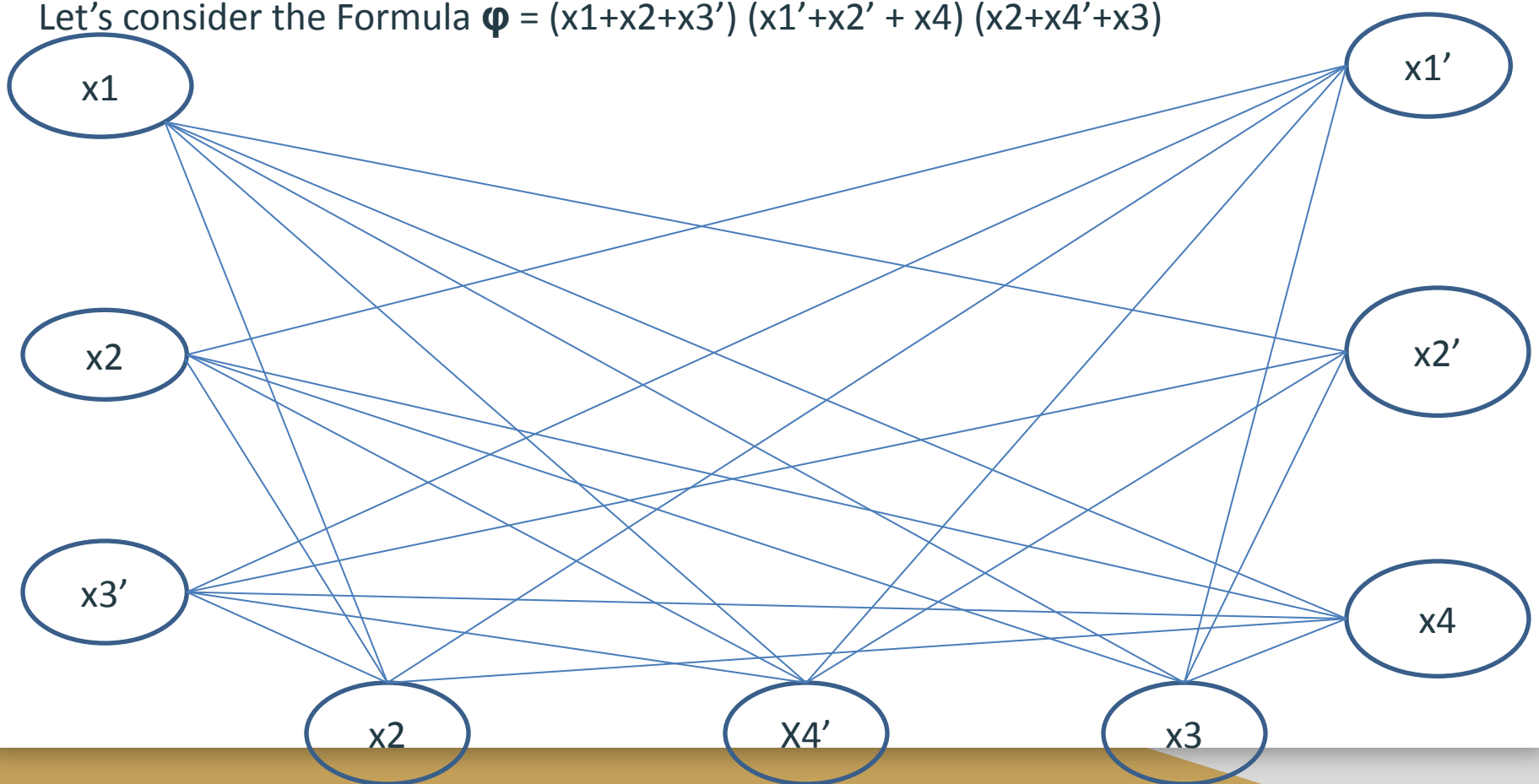
Let's consider the Formula F = (x1+x2+x3') (x1'+x2' + x4)

The graph has a 2 – Clique.
Consider x1 = True and x4 = True



3-SAT $\leq_R$ K-Clique (Example -2)

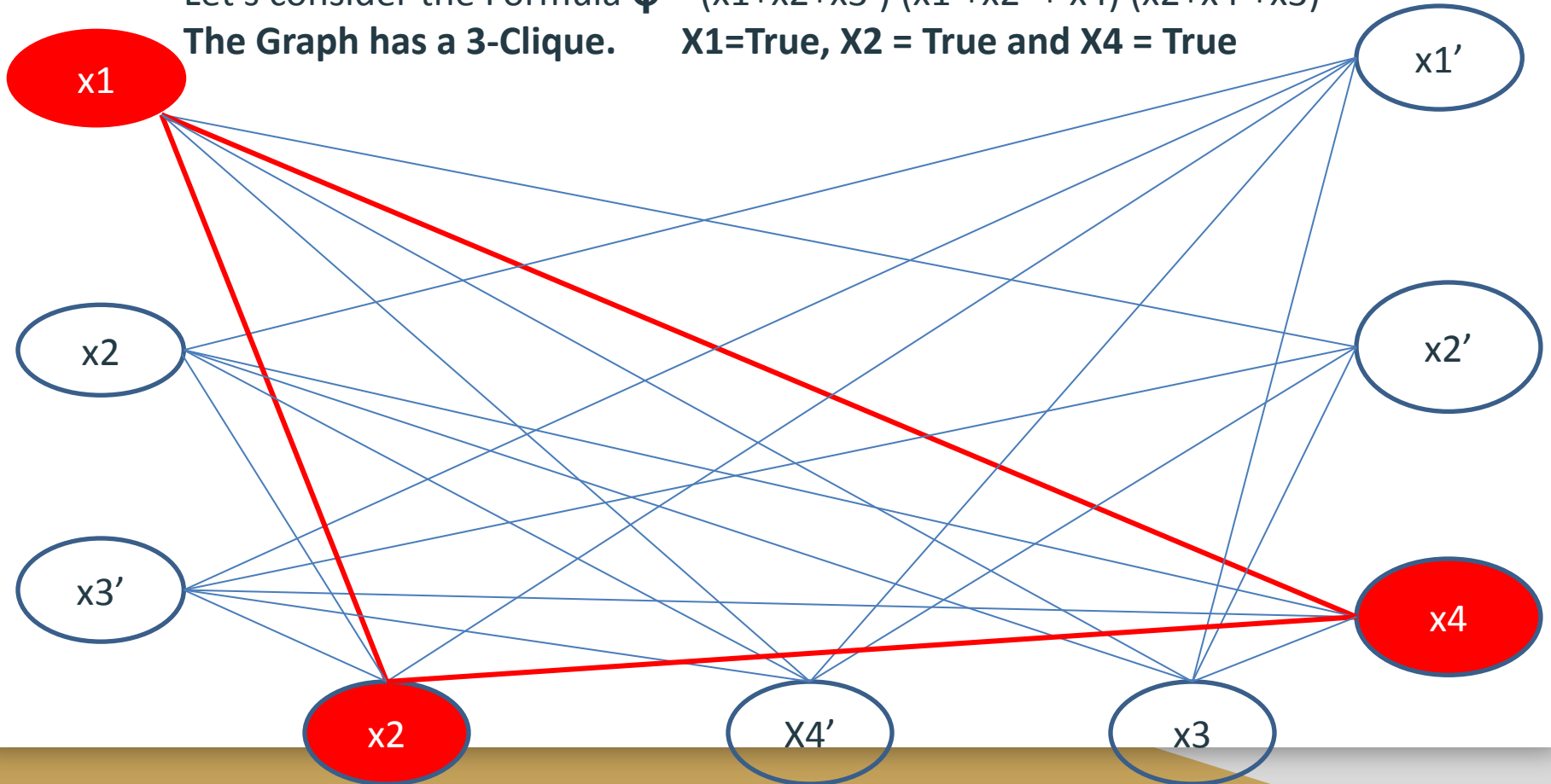
Let's consider the Formula **φ** = (x1+x2+x3') (x1'+x2' + x4) (x2+x4'+x3)



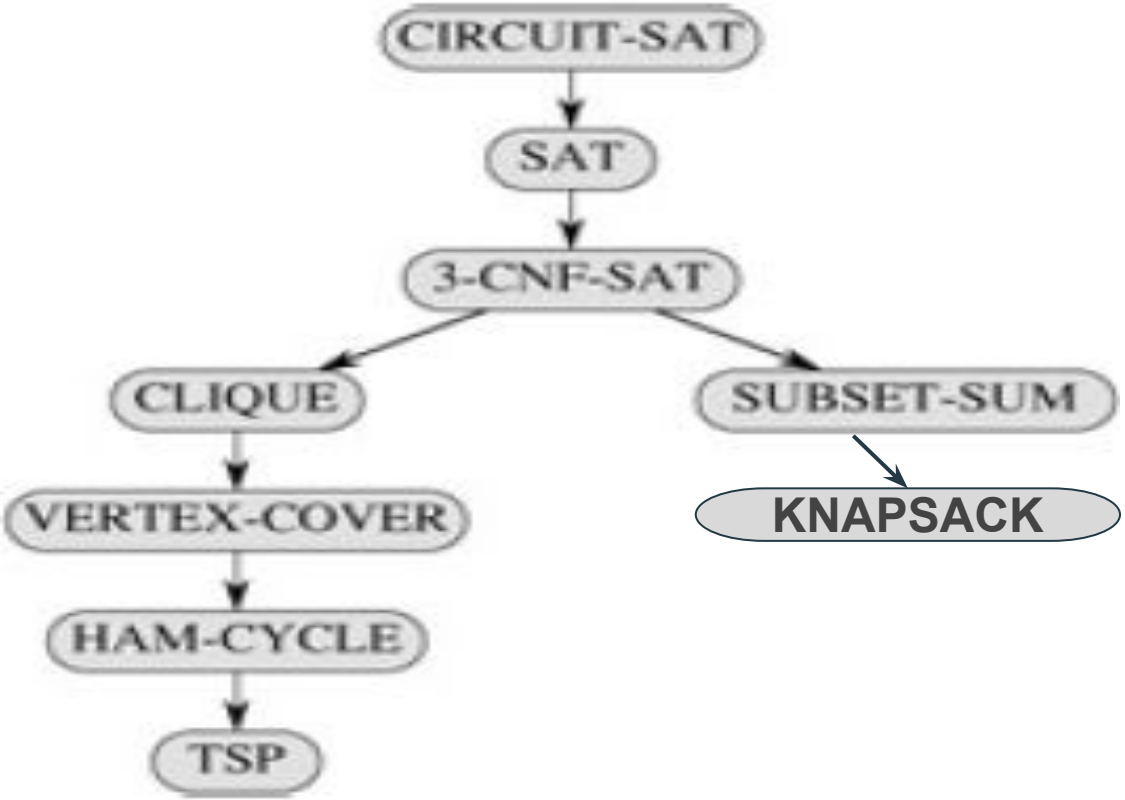
3-SAT $\leq_R$ K- Clique (Example -2)

Let's consider the Formula **φ** = (x1+x2+x3') (x1'+x2' + x4) (x2+x4'+x3)

The Graph has a 3-Clique. **X1=True, X2 = True and X4 = True**



Structure of Proof



Subset Sum $\leq_R$ 0/1 Knapsack Problem

2.1 subset sum problem

Given a set S of n positive integers, and a positive value SUM, does there exist a subset S' of S for which sum of all elements $\sum S'_i = SUM$?

Input : $S = \{3, 2, 7, 1\}$, $SUM = 6$

Output: $True$, subset is $S' = \{3, 2, 1\}$

2.2 0/1 Knapsack Problem

Given a set I of n items, with corresponding profit values P, weight values W, Capacity C and max-profit V; does there exists a subset I' of items I for which total weight $\sum W'_i \leq C$ and total profit $\sum P'_i \geq V$?

Input : $I = \{i_1, i_2, i_3, i_4\}$, $P = \{4, 2, 6, 8\}$, $W = \{2, 3, 1, 2\}$, $C = 5$, $V = 14$

Output : $True$, $I' = \{i_3, i_4\}$

Reduction Procedure

a. **Take an instance of subset sum :** $S = \{3, 2, 7, 1\}$, $SUM = 6$

b. **Define a procedure to create an instance of 0/1 knapsack using the instance of subset sum:**

$|I| = n$, where n = no. of elements in S.

$V = SUM$

$C = SUM$

$P = S$

$W = S$

so, $I = \{i_1, i_2, i_3, i_4\}$, $P = \{3, 2, 7, 1\}$, $W = \{3, 2, 7, 1\}$, $V = 6$, $C = 6$

c. **Solve 0/1 knapsack problem:** To prove the reduction we assume that there exist a polynomial time algorithm to solve 0/1 knapsack problem. But here to write the program and to understand the procedure, we shall use dynamic programming algorithm to solve 0/1 knapsack. The solution for the instance given in step b is "True" i.e. there is a subset of items for which total profit = 6, and total weight = 6.

d. **Find solution of subset sum from the solution of 0/1 knapsack :**

$Solution(subsetsum) = solution(0/1knapsack)$

$Solution(subsetsum) = True$

e. **Show that** b **and** d **takes polynomial time :** Step b and d are assignments of finite size, hence both can be done in constant time.

Reduction Procedure

Algorithm 1 Algo_subsetsum $(S = \{s_1, s_2, ...s_n\}, SUM)$

$P \leftarrow S$
$W \leftarrow S$
$C \leftarrow SUM$
$V \leftarrow SUM$
if Algo_dynamic_knapsack(P, W, C, V) == "TRUE" **then**
  RETURN "TRUE"
else
  RETURN "FALSE"
end if

Algorithm 2 Algo_dynamic_knapsack (P, W, C, V)

Create $M[0...N][0...C]$ //NXC dimension matrix to store solutions of sub problems
while w from 0 to C **do**
  $M[0, w] \leftarrow 0$
end while
while i from 1 to n **do**
  $M[i, 0] = 0$
end while
while i from 1 to n **do**
  while w from 0 to C **do**
    if $W_i \leq w$ **then**
      if $P_i + M[i-1, w-W_i] > M[i-1, w]$ **then**
        $M[i, w] \leftarrow P_i + M[i-1, w-W_i]$
      else
        $M[i, w] \leftarrow M[i-1, w]$
      end if
    else
      $M[i, w] \leftarrow M[i-1, w]$
    end if
  end while
end while
if $M[N, W] \geq V$ **then**
  RETURN "TRUE"
else
  RETURN "FALSE"
end if

NP-Hard Problem which is not NP-Complete

Unfortunately, there are many computing problems for which the best possible algorithm takes a long time to run. A simple example is the ***Towers of Hanoi problem*** which requires 2^n moves to "solve" a tower with n disks.

NP-Hard Problem which is not NP-Complete

It is not possible for any computer program that solves the Towers of Hanoi problem to run in less than $\Omega(2^n)$ time, because that many moves must be printed out.

Besides those problems whose solutions *must* take a long time to run, **there are also many problems for which we simply do not know if there are efficient algorithms or not.**

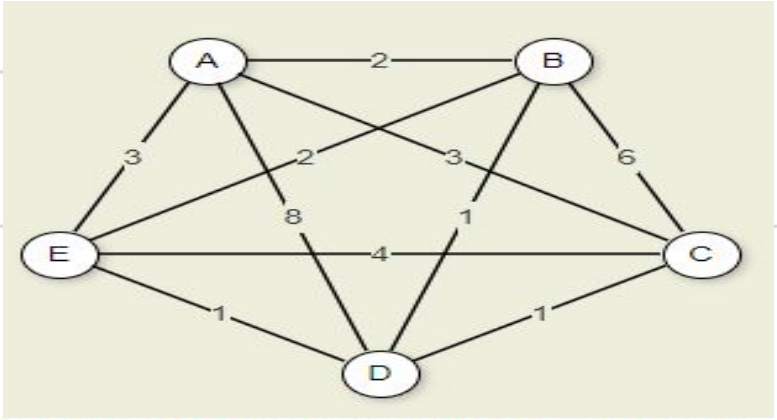
Optimization Problem Vs. Decision Problem

Problem

TRAVELING SALESMAN (1)

Input: A complete, directed graph G with positive distances assigned to each edge in the graph.

Output: The shortest simple cycle that includes every vertex.



Problem

TRAVELING SALESMAN (2)

Input: A complete, directed graph G with positive distances assigned to each edge in the graph, and an integer k.

Output: YES if there is a simple cycle with total distance $\leq k$ containing every vertex in G, and NO otherwise.

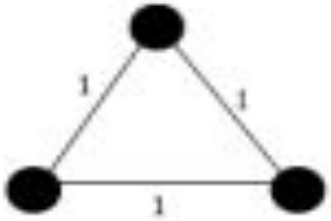
Decision Problems which are not NP-Complete

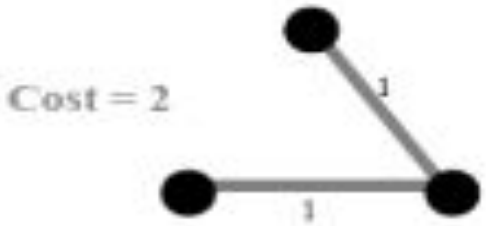
There are decision problems that are NP-hard but not NP-complete such as the **halting problem**. That is the problem which asks "given a program and its input, will it run forever?"

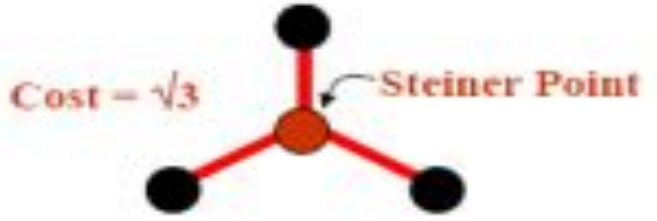
That is a yes/no question and so is a decision problem.

Minimum Steiner Tree Problem (NP-Hard)

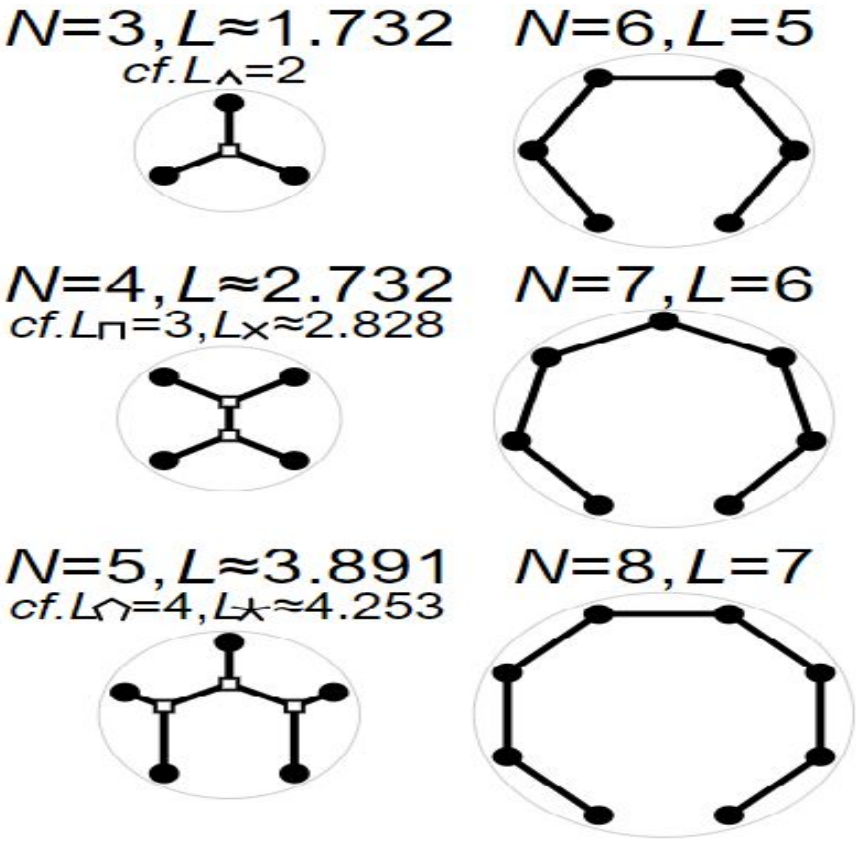
Given an undirected graph with non-negative edge weights and a subset of vertices, usually referred to as **terminals,** the Steiner tree problem in graphs requires a tree of minimum weight that contains all terminals (but may include additional vertices) and minimizes the total weight of its edges.



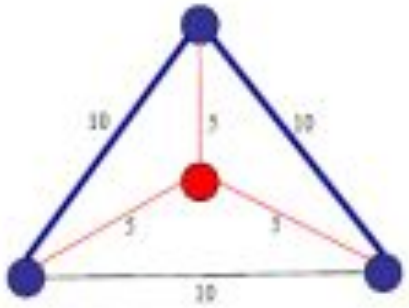




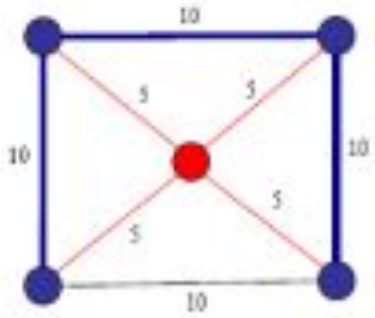
Minimum Steiner Tree Problem (NP-Hard)



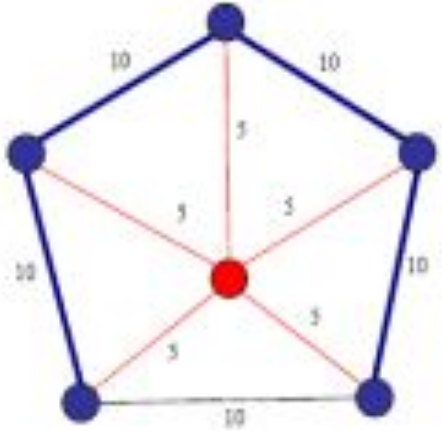
Minimum Steiner Tree Problem (NP-Hard)



MST: 20
Steiner Tree: 15
Ratio: 1.33



MST: 30
Steiner Tree: 20
Ratio: 1.5



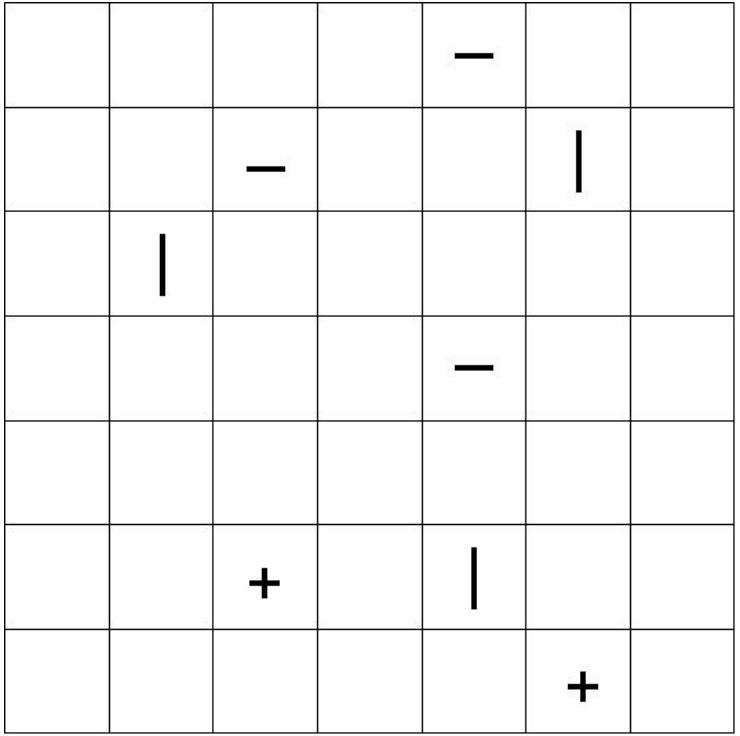
MST: 40
Steiner Tree: 25
Ratio: 1.8

Minimum Steiner Tree Problem (NP-Hard)

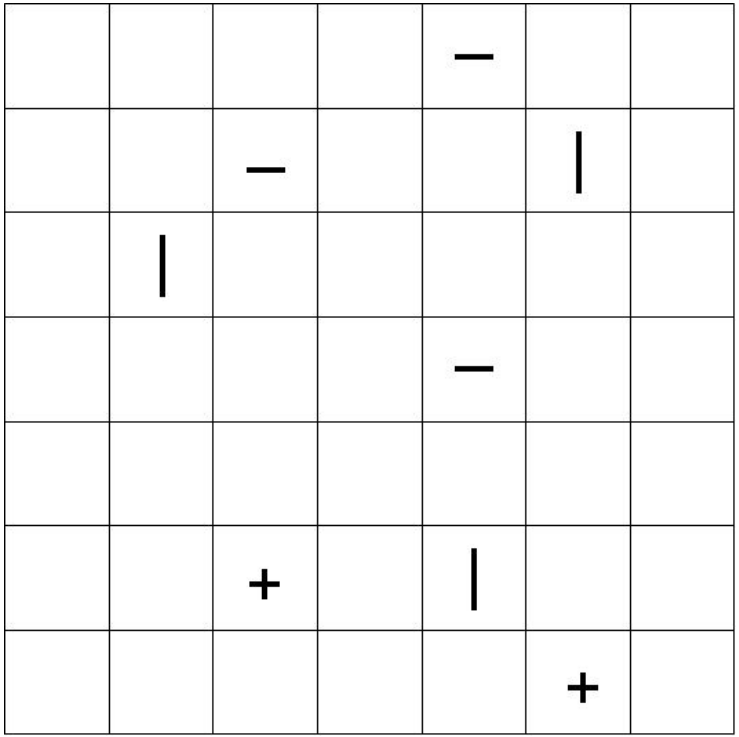
Its decision variant, asking whether a given input has a tree of weight less than some given threshold, is NP-complete, which implies that the optimization variant, asking for the minimum-weight tree in a given graph, is NP-hard. In fact, the decision variant was among Karp's original 21 NP-complete problems.

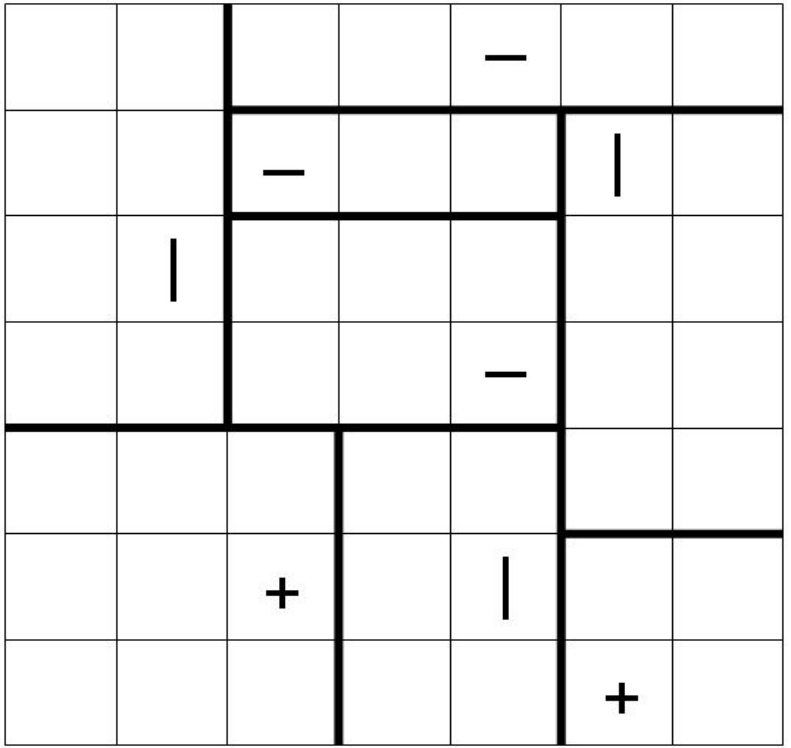
Tatamibari puzzle (NP-Complete)

- Every partition must contain exactly one symbol in it.
- A + symbol must be contained in a square.
- A | symbol must be contained in a rectangle with a greater height than width.
- A - symbol must be contained in a rectangle with a greater width than height.
- Four pieces may never share the same corner.



Tatamibari puzzle (NP-Complete)





By Matanlevine - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=27561076

NP-Hard	NP-Complete
NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) that can be reducible into X in polynomial time.	NP-Complete problems can be solved by a non-deterministic Algorithm n polynomial time
To solve this problem, it do not have to be in NP .	To solve this problem, it must be both NP and NP-hard problems.
Not all NP-hard problems are NP-complete.	All NP-complete problems are NP-hard
Do not have to be a Decision problem.	It is exclusively a Decision problem.
It is mostly an Optimization Problem	It is mostly a Decision Problem.
Example: Halting problem, Tower of Hanoi, Minimum Vertex cover problem, etc.	Example: Determine whether a graph has a Hamiltonian cycle, Determine whether a Boolean formula is satisfiable or not, Vertex cover of size k exists or not, etc.

Thank You